Interfacing Engines and Schedulers in Or-Parallel Prolog Systems

Péter Szeredi*, Rong Yang
Department of Computer Science
University of Bristol, Bristol BS8 1TR, U.K.

Mats Carlsson
Swedish Institute of Computer Science
P.O. Box 1263, S-164 28 Kista, Sweden

January 4, 1995

Abstract

Parallel Prolog systems consist, at least conceptually, of two components: an engine and a scheduler. This paper addresses the problem of defining a clean interface between these components. Such an interface has been designed for Aurora, a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors.

The practical purpose of the interface is to enable different engine and scheduler implementations to be used interchangeably. The development of the interface has, however, contributed in great extent to the clarification of issues in exploiting or-parallelism in Prolog. We believe that these issues are relevant to a wider circle of research in the area of or-parallel implementations of logic programming.

We believe that the concept of an engine-scheduler interface is applicable to a wider range of parallel Prolog implementations. Indeed, the present interface has been used in the Andorra-I system, which supports both and- and or-parallelism.

Keywords: Or-Parallel Execution, Multiprocessors, Implementation Techniques, Scheduling.

1 Introduction

Parallel Prolog systems consist, at least conceptually, of two components: an *engine*, which is responsible for the actual execution of the Prolog code, and a *scheduler*, which provides the engine component with work. This paper addresses the problem of defining a clean interface between these components. We focus on a particular interface which has evolved within the implementation of an or-parallel Prolog system, Aurora. The interface has successfully been used to connect the Aurora engine with four different schedulers. It has subsequently been applied in the implementation of the and-or-parallel language Andorra-I, thus proving that its generality extends beyond or-parallel Prolog.

Aurora is a prototype or-parallel implementation of the full Prolog language for shared memory multiprocessors, based on the SRI model of execution [16], and currently running on Sequent and Encore machines. It has been developed in the framework of the Gigalips project [11], a collaborative effort between groups at the Argonne National Laboratory in Illinois, the University of Bristol (previously at the University of Manchester) and the Swedish Institute of Computer Science (SICS) in Stockholm.

The issue of defining a clear interface between the engine and scheduler components of Aurora was raised in the early stages of the implementation effort. Ross Overbeek made the first attempt to

* *On leave from (and present address)* SZKI IQSOFT, Donáti u. 35-45, Budapest, Hungary.

formulate such an interface and Alan Calderwood produced the version [7] used in the first generation of Aurora (based on SICStus Prolog version 0.3).

A fundamental revision of the interface was necessitated by several factors. Performance analysis work on Aurora [14] has shown that some unnecessary overheads are caused by design decisions enforced by the interface. Development of new schedulers and extensions to existing algorithms required the interface to be made more general. The Aurora engine has also been rebuilt on the basis of SICStus Prolog version 0.6.

The new interface, described in the present paper, is part of the second generation of Aurora. The major changes with respect to the previous interface are the following:

- execution is governed by the engine, rather than the scheduler;
- the set of basic concepts has been made simpler and more uniform;
- several potential optimisations are supported;
- the interface is extended to support transfer of information related to pruning operators [10].

The paper is organised as follows. Section 2 summarises the SRI model and defines the necessary concepts. Section 3 gives a top level view of the interface. Section 4 presents the data structures involved in the interface, while Sections 5 and 6 describe engine-scheduler interactions in various phases of Aurora execution. Section 7 shows the extensions: handling of pruning information and various optimisations. Section 8 discusses the major issues involved in implementing the Aurora engine side of the interface. Section 9 describes how the interface was utilised to introduce or-parallelism into the Andorra-I system [12]. Section 10 presents preliminary performance results from the Aurora implementation. We end with a short concluding section.

A complete description of the interface is contained in [15].

2 Preliminaries

Aurora is based on the SRI model [16]. According to this model the system consists of several *workers* (processes) exploring the search tree of a Prolog program in parallel. Each node of the tree corresponds to a Prolog *choicepoint* with a branch associated with each alternative clause. A predicate can optionally be declared *sequential* by the user, to prohibit parallel exploration of alternative clauses of a predicate. Corresponding nodes are also annotated as sequential. All other nodes are *parallel*.

As the tree is being explored, each node can be either *live*, i.e. have at least one unexplored alternative, or *dead*. A node is a *fork node* if there are two or more branches below it; otherwise, it is a *nonfork node*. A fork node cannot be sequential. Live parallel nodes, and live sequential nodes with no branches below them, correspond to tasks that can be executed by workers. Each worker has to perform activities of two basic types:

- executing the actual Prolog code;
- finding work in the tree, providing other workers with work and synchronising with other workers.

In accordance with the SRI model each worker has a separate *binding array*, in which it stores its own bindings to potentially shared variables (conditional bindings). This technique allows constant time access to the value of a shared variable, but imposes an overhead of updating the binding arrays whenever a worker has to move within the search tree.

The or-tree is divided into an upper, *public*, part accessible to all workers and a lower, *private*, part accessible to only one worker. A worker exploring its private region does not have to be concerned with synchronisation or maintaining scheduling data; it can work very much like a standard Prolog engine. The boundary between the public and private regions changes dynamically. It is one of the critical aspects of the scheduling algorithm to decide when to make a node public, allowing other workers to

share work at it. In the majority of schedulers, the worker will make his *sentry* node, i.e. his topmost private node, public when all nodes above it have become dead, i.e. have no more alternatives to explore. This means that each worker tries to keep a piece of work on its branch available to other workers.

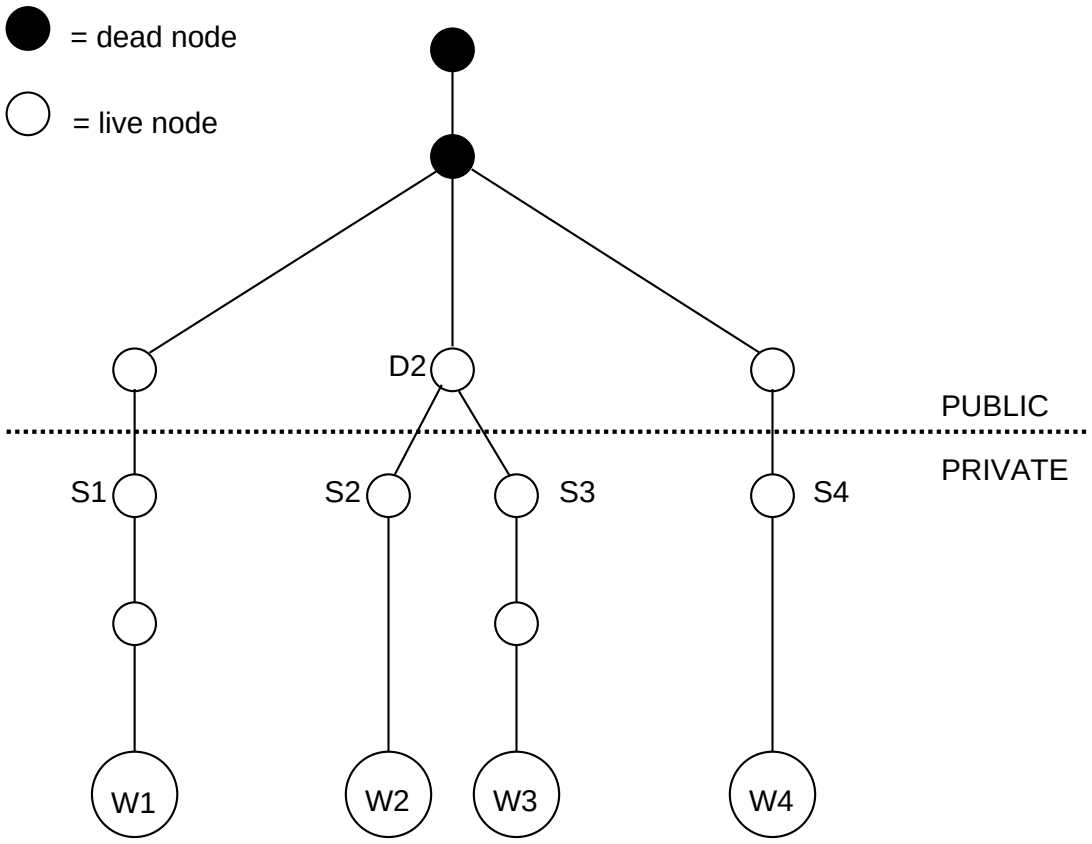

Figure 1: THE OR-TREE OF THE SRI MODEL

For example, in Figure 1, an or-tree being explored by four workers (W1–W4) is shown. The workers' sentry nodes are denoted S1–S4. Assume that there is an unexplored alternative at node D2. Now if the branch being explored by worker W1 dies back and W1 takes the alternative at D2, the node D2 will become dead, and the scheduler will normally extend the public region to include nodes S2–S3 so as to keep a piece of work available on every branch.

The exploration by a worker of its private region constitutes that worker's *assignment*, which normally terminates if the worker backtracks into the public part. The assignment terminates prematurely if the branch is *suspended*, or if it is *pruned* by some other worker.

There are three *pruning operators* currently supported by Aurora: the conventional Prolog *cut*, which prunes all branches to its right and a symmetric version of cut called *commit*, which prunes branches both to its left and right. A cut or a commit must not, and will not, go ahead if there is a chance of being pruned by a cut with a smaller scope. The third type of pruning operator is the *cavalier commit* which is executed immediately, even if endangered by a smaller cut. The cavalier commit is provided for experimental purposes only, it is expected to be used in exceptional circumstances, for operations similar to `abort` in Prolog. Work done in the scope of a pruning operator is said to be *speculative*.

Suspension is used to preserve the observable semantics of Prolog programs executed by Aurora: when a built-in predicate with some side-effect is reached on a non-leftmost branch of the search tree, or when a pruning operator is reached on a branch which could be pruned by a cut with a smaller scope, the execution must be suspended. Furthermore the scheduler may decide to suspend the current

branch when less speculative work can be done somewhere else in the tree.

Four separate schedulers are currently being developed for Aurora. The Argonne scheduler [6] relies on data stored in the tree itself to implement a local strategy according to which live nodes "attract" workers without work. When several workers are idle they will compete to get to a given piece of work and the fastest one will win. The Manchester scheduler [8] tries to select the nearest worker in advance, without moving over the tree. It uses global data structures to store some of the information on available work and workers. The wavefront scheduler [5] uses a special distributed data structure, the *wavefront*, to facilitate allocation of work to workers. The Bristol scheduler [3] tries to minimise scheduler overhead by extending the public region eagerly: sequences of nodes are made public instead of single nodes, and work is taken from the bottommost live node of a branch.

3 The Top Level View of the Interface

The principal duty of the scheduler is to provide the engine with work. The thread of control thus alternates between the two components: the engine executes a piece of Prolog code, then the scheduler finds the next assignment, passes control back to the engine, etc. A possible way of implementing this interaction is to put the scheduler *above* the engine: the scheduler *calls* the engine when it finds a suitable piece of work to be executed and the engine *returns* when such an assignment has been finished. In fact this scheme was the basis of earlier interfaces in Aurora [7].

We use a different approach in the current version of Aurora. The execution is governed by the engine: whenever it finishes an assignment, it calls an appropriate scheduler function to provide a new piece of work. The advantage of this scheme is that the environment for Prolog execution (e.g. the set of WAM-registers) is not destroyed when an assignment is terminated and need not be rebuilt upon returning to work. This is of special importance for Prolog programs with fine granularity (i.e. small assignment size), where switching between engine and scheduler code is very frequent [14].

Figure 2 shows the top view of the current interface. This is centered around the engine doing work. All the other boxes in the picture represent scheduler functions called by the engine. Note the convention that the names of all scheduler functions are prefixed with '`Sched_`'.

The functions shown in Figure 2 are arranged in three groups:

- finding work (left side of Figure 2);
- communication with other workers during work (lower part of Figure 2), e.g. when cuts or side effect predicates are to be executed;
- certain events during work that may be of interest to the scheduler (right side of Figure 2), e.g. creation and destruction of nodes.

The four boxes on the left of Figure 2 represent the so called *functions for finding work*:

`Sched_Start_Work` is used to acquire work for the first time, immediately after the initialisation of the worker;

`Sched_Die_Back` is called when the engine backtracks to a public node;

`Sched_Be_Pruned` is invoked when the worker's current branch is pruned off by another worker;

`Sched_Suspend` is called when the worker has to suspend its current branch.

These functions differ in their initial activities, but normally continue with a common algorithm for "looking for work" (see Section 5). This algorithm has two possible outcomes: either work is found, or the whole system is halted. Correspondingly each of the functions for finding work has two exits: the normal one (shown on the right side of the function boxes in Figure 2) leads back to work, while the other exit (left hand side) leads to the termination of the whole Aurora invocation.

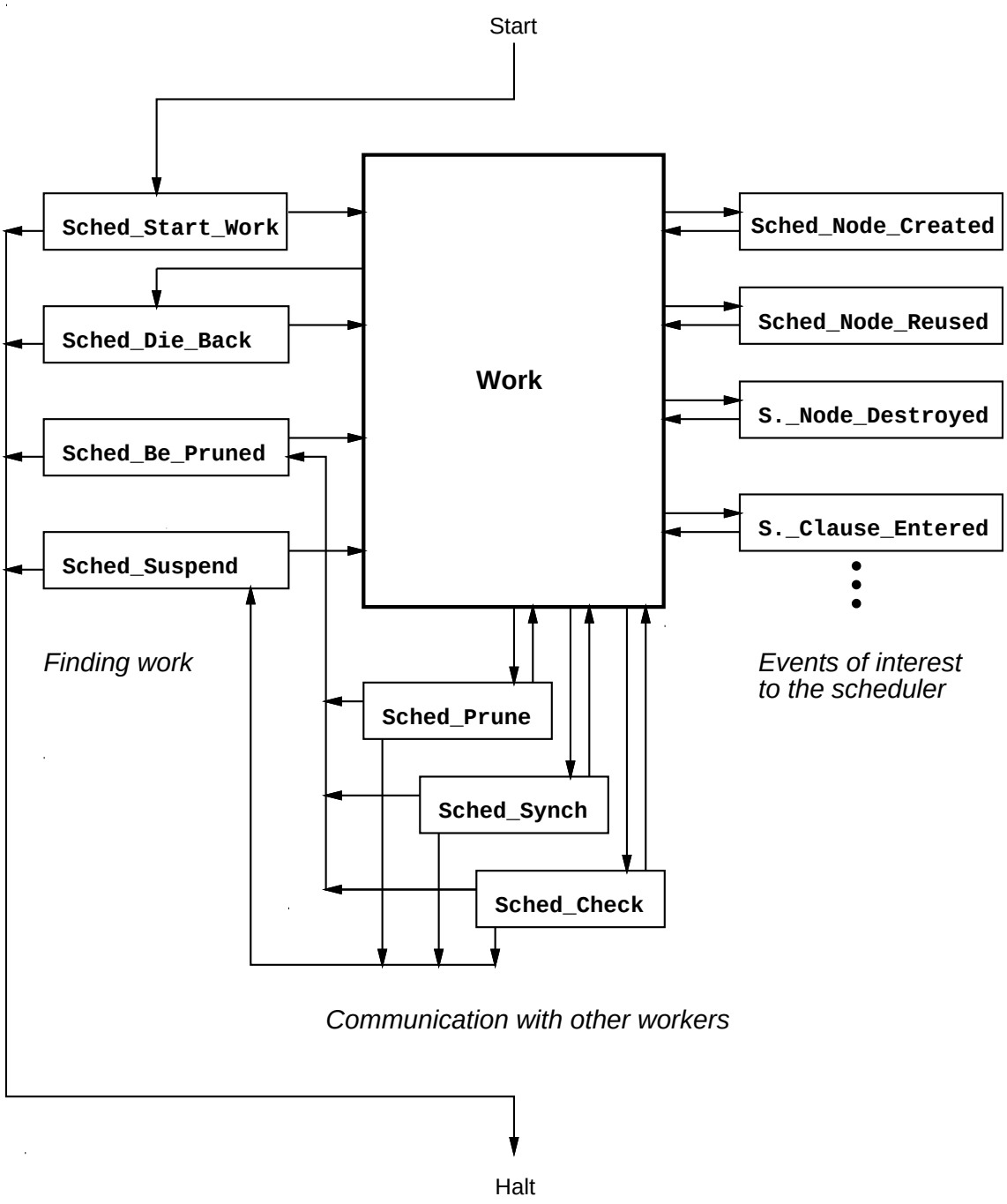

Figure 2: The top level view of the interface

The next group of interface functions provided by the scheduler is depicted at the bottom of Figure 2. These functions are called during work, when the engine may require some assistance from the scheduler (mainly in order to communicate with other workers):

`Sched_Prune` — when a cut or commit is executed;

`Sched_Synch` — when a predicate with side effects is encountered;

`Sched_Check` — at every Prolog procedure call (to check for interrupts).

The above functions have three exits. The normal exit (depicted by upwards arrows in Figure 2) leads back to work. The other two exits correspond to premature termination of the current assignment, when the current branch has been pruned or has to suspend (leftward and downward arrows). In both cases the engine will do the housekeeping operations necessary for the given type of assignment termination, and proceed to call the scheduler to find the next assignment. See Section 6 for a more detailed description of the functions for communication with other workers.

The third group of functions shown in Figure 2 (right hand side) corresponds to some events during work that may be of interest to the scheduler. A common property of this group is that the interface does not prescribe any specific activity to be done by these functions: the scheduler is merely given an opportunity to do whatever is needed for maintaining its data structures. As an example, `Sched_Node_Created` (and the corresponding `Sched_Node_Destroyed`) can be used to keep track of the presence of parallel nodes in the private region—as a prospective source of work for other workers. Similarly `Sched_Clause_Entered` can be utilised for maintaining information about the presence of pruning operators in the current branch (see Section 7.2).

There are further groups of scheduler functions, not shown in Figure 2. These are used in the initialisation of the whole system, in handling keyboard interrupts , and in the implementation of certain optimisations (Section 7.1).

The engine side of the interface consists of several groups of functions that support the scheduler algorithm:

- providing access to certain data structures (nodes and alternatives) maintained by the engine,
- extending the public region on the current branch of execution,
- positioning the engine (i.e. the binding array) in the search tree, while looking for work,
- notifying the engine of certain events, e.g. work being found.

The data structure aspects of the engine interface are presented in Section 4. Other interface functions provided by the engine will be described in Sections 5 and 6.

4 Common Data Structures

The engine is responsible for maintaining the *node stack*, a principal data area of major importance to the scheduler. The engine defines the *node* data type, but the scheduler is expected to supply a number of fields to be included in this structure for its own purposes.

Among the node fields defined by the engine, some are of interest to the scheduler. Access functions for these fields are provided in the interface:

`Node_Level` — the distance of the node from the root of the search tree,

`Node_Parent` — a pointer to the parent node in the tree,

`Node_Alternatives` — a pointer to the next unexplored alternative of the node.

The scheduler-specific fields of the node data structure normally include pointers describing the topology of the tree. For example, most schedulers will have fields storing a pointer to the first child and the next sibling of a node.

An additional common static data structure, the *alternative*, is introduced to allow the schedulers to keep static data related to clauses. This data structure is used in the Aurora engine to replace the `'try'`, `'retry'` and `'trust'` instructions of WAM [9]. Each clause of the user program is represented by an alternative, which stores a pointer to the code of the clause and a pointer to the successor alternative, if any. If a predicate is subject to indexing, the compiler may create several chains of alternatives to cater for different values in the indexing argument position. This means that several alternatives can refer to the same clause.

The scheduler may supply a number of fields to be included in the alternative structure, to accommodate any (static) information to be associated with clauses. The scheduler can derive this data from the information supplied by the engine when alternatives are created (`Sched_Alternative_Created`). There are two types of static data supplied by the engine:

- information about sequential predicates—this information is normally stored in each alternative of the predicate.
- pruning information—data on the number of pruning operators (cuts, commits and conditional expressions) contained in the clause or the predicate (see Section 7.2).

The only engine field in the alternative structure that is of interest to the scheduler is the one pointing to the successor alternative (`Alternative_Next`). This field is used, for example, when the scheduler starts a new branch from a public node and needs to advance the next alternative pointer of the node.

5 Finding Work

Figure 3 shows the engine functions used by the scheduler while it is looking for work. The actual algorithms of the four functions for finding work will normally differ, but they all use the same set of engine support functions.

Functions `Move_Engine_Up` and `Move_Engine_Down`, shown on the right hand side of Figure 3, instruct the engine to move the binding array up or down the current branch. Initially, the binding array is positioned at or below the youngest public node on the branch. Before returning, the scheduler has to position the binding array above the new sentry node.

Different schedulers employ different strategies in moving over the tree. The Argonne scheduler moves node-by-node, when approaching the potential work node. Other schedulers locate a piece of work from a distance and move the engine to the appropriate place in a few big jumps.

There is no need to move the engine if work is taken from the parent of the old sentry node. An additional entry point to the scheduler, `Sched_Get_Work_At_Parent` (see Section 7.1), has been provided for this special case.

The left hand side of Figure 3 shows the engine functions for memory management of the node stack. A worker may have to remove some dead nodes from the tree as it moves upwards. This involves deleting these nodes from the scheduler data structures (normally the sibling chain) and invoking the `Mark_Node_Reclaimable` engine function. As a special case, the old sentry node will have to be deleted from the tree at the beginning of `Sched_Die_Back` and `Sched_Be_Pruned`.

When the scheduler decides to reserve a new piece of work from a live public node (work node), it has to create a sentry node for the new branch. This involves calling the `Allocate_Node` function, which first removes all the nodes that have been marked as reclaimable from the top of the worker's stack and then allocates a new sentry node. The related `Allocate_Foreign_Node` function is used if *another* worker allocates a node on the stack of the worker looking for work. This is used in the Manchester scheduler to implement handing work to an idle worker.

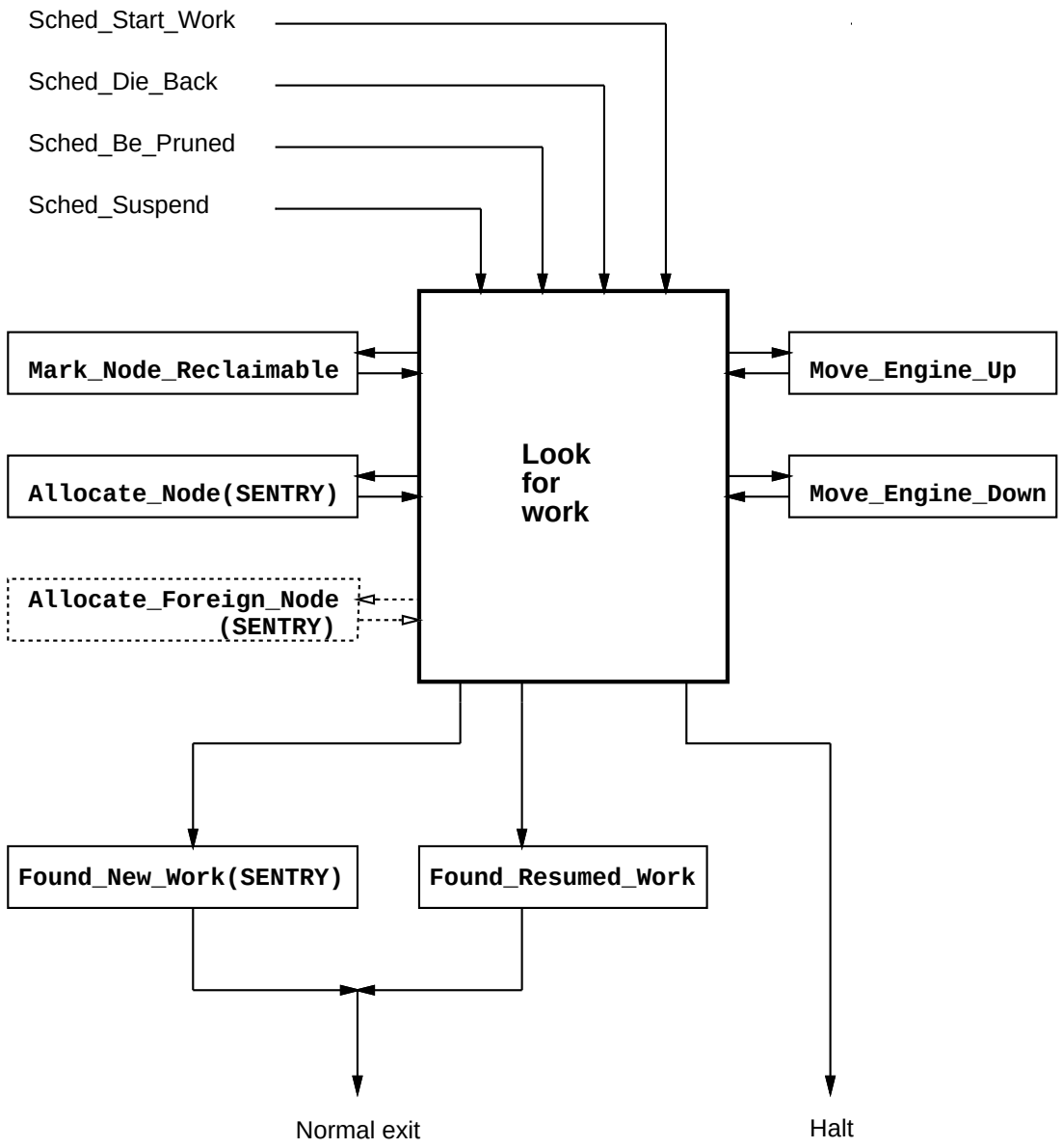

Figure 3: Engine functions in looking for work

The new sentry node serves as a placeholder for the new assignment. The scheduler inserts the sentry into the search tree and simultaneously reserves an alternative to be explored by the new branch (by reading and advancing the `Node_Alternatives` field of the work node).

The bottom part of Figure 3 shows the possible exit paths from the functions for finding work. The actual work found can correspond either to a new branch or to a branch which was hitherto suspended and can be resumed now. Functions `Found_New_Work` and `Found_Resumed_Work` are used to notify the engine about the type of the work found, and to supply the new sentry node. The box for `Found_New_Work` in Figure 3 shows the `SENTRY` argument to highlight the fact that this argument should be the same as the one returned in `Allocate_...Node`.

6 Communication with Other Workers

The need for communication with other workers arises when a pruning operator or a built-in predicate with side effects is to be executed. In addition, a periodic check is needed to examine if there are communication requests from other workers.

The `Sched_Prune` function is invoked when a pruning operator is encountered. At this moment the engine has already executed the private part of the pruning. The scheduler receives a pointer to the cut node (showing the scope of pruning) and an argument indicating the type of the pruning operator (cut, commit or cavalier commit). It has to check if the preconditions for pruning are satisfied: the current branch should not be pruned itself, and, except for the cavalier commit, it should not be endangered by cuts with a smaller scope, as discussed in [10]. The latter condition can be replaced by a requirement for the branch to be leftmost in the subtree rooted at the child of the cut node, if the scheduler does not maintain specific pruning information.

If the preconditions of pruning are not satisfied, `Sched_Prune` uses one of the abnormal exits (cf. Figure 2) to indicate that the branch has been killed or that it has to suspend (waiting to become leftmost). If the pruning operation can go ahead, the scheduler has to locate the workers that are in the pruned subtree and interrupt them. There may be branches in this subtree which have previously been suspended. A special engine function, `Mark_Suspended_Branch_Reclaimable`, is used for cleaning up such branches.

The `Sched_Synch` function is invoked when a call to a built-in predicate with side-effects is encountered. Normally such calls are executed only when their branch becomes leftmost in the whole tree. There are, however, some special predicates (e.g. those used to assert solutions in a `setof`), for which the order of invocation is not significant: their execution can go ahead if not endangered by a cut within a specific subtree. The `Sched_Synch` function receives an argument encoding the type of the check needed, and a pointer to the root of the subtree concerned.

The third communication function, `Sched_Check`, is called at every Prolog procedure call. Frequent invocation of this function is necessary so that the scheduler can answer requests (e.g. interrupts) from other workers without too much delay. Note, however, that a scheduler may choose to do the checks only after a certain number of `Sched_Check` invocations (as is the case for the Manchester and Argonne schedulers).

The nature of requests to be handled by `Sched_Check` varies from scheduler to scheduler. There are, however, two common sets of circumstances: the worker may be requested to kill its assignment or to make some of its private nodes public (to make work available to other workers). The latter activity needs assistance from the engine: the function `Make_Public` extends the public region on the current branch down to a specified node.

7 Extensions of the Basic Interface

7.1 Simplified Backtracking

When a worker backtracks to a live public node and is able to take a new branch from there, several administrative activities can be avoided. The sentry node can be re-used, rather than being marked

as reclaimable and re-allocated. There is scope for a related optimisation in the scheduler: instead of deleting the old sentry from the sibling chain and then installing it as the last sibling, the scheduler can move the sentry node to the end of the sibling chain (or do nothing if the old sentry was the last child). The interface supports this important optimisation by a function `Sched_Get_Work_At_Parent`, called when the engine backtracks to a live public node. If the scheduler, following the necessary synchronisation operations, still finds the node to be live, it can reserve an alternative from that node. If the scheduler cannot take work from the node in question, it returns to the engine, which will subsequently invoke `Sched_Die_Back` to acquire a new piece of work.

The `Sched_Get_Work_At_Parent` function also supports the *contraction* operation of the SRI model [16]. This operation removes a dead nonfork node after the last alternative has been taken from it. The node in question can be physically removed only if it is on the top of the stack of the worker executing the given branch.

7.2 Pruning Information

Information about the presence of pruning operators in a clause may be needed by the scheduler to perform pruning more efficiently or to distinguish between speculative and non-speculative work. Various algorithms related to pruning have been developed and discussed in [10]. When designing the interface, we tried to generalise and extend the format of pruning data as described in [10], so that other possible approaches (e.g. [13]) can be supported as well.

If one disregards disjunctions, the information needed about pruning is quite simple. A scheduler may wish to know whether a clause contains cuts or commits[1]. For more exact pruning algorithms the number of occurrences of each pruning operator may be needed. The fact that a clause must fail, may also be of interest: when such a clause is entered, the pruning operators in the current continuation (i.e. in the previous resolvent) become inaccessible. The simple set of pruning data would thus consist of three items for each clause: the number of cuts, the number of commits and the Boolean value indicating whether the clause ends in a failing call (i.e. `fail`, but in the future, global compile time analysis might discover this property for other calls).

The presence of disjunctions and conditionals makes the situation more complicated. In [15] we present a set of pruning data consisting of seven items, to describe the pruning properties of a general clause (one that may contain disjunctions and conditionals).

8 Implementation of the Interface in the Aurora Engine

The Aurora emulator [9] was produced by modifying the SICStus emulator to support the SRI model and by converting it from a stand-alone program to an Aurora worker component connected by an algorithmic interface to a scheduler component. The total performance degradation resulting from these changes has been found to be around 25%. In an earlier paper [11] we gave an overview of the changes imposed by the SRI model. In this section we concentrate on the impacts of the interface on the engine and on changes introduced in the new design.

8.1 Boundaries

The engine needs to maintain the boundary between the public and private regions. Within the private region, it must distinguish between *local* nodes, i.e. nodes adjacent to the top of the worker's own stack, and remote nodes. This is achieved by storing a pointer to the respective boundary nodes in certain registers. These registers are initialised when an assignment is started (`Found...Work`). They are updated when the public region is extended (`Make_Public`) or contracted (`Sched_Get_Work_At_Parent`), and when backtracking in the private region winds back to the worker's own stack. They are consulted to distinguish different cases of backtracking and pruning operations.

[1] Note that data on cavalier commits is not included in the pruning information, as this operation is expected to be used only for handling exceptional circumstances.

8.2 Backtracking

From the engine's point of view, the main complication of or-parallel execution is its impact on the backtracking routine. This routine has to check whether it is about to backtrack into the public region, in which case the scheduler must be invoked to perform public backtracking (`Sched_Die_Back` or `Sched_Get_Work_At_Parent`). Private backtracking has to face the complication that the private region may extend to other workers' stacks, and possibly wind back to the worker's own back again. As explained earlier, remote nodes cannot be reclaimed when they are trusted; instead, `Mark_Node_Reclaimable` is invoked when dying back over a remote node.

Shallow backtracking is optimised in the private region, but only if the current node is on the top of the worker's own stack.

8.3 Memory Management

As stated earlier, the stack memory management relies on the node stack. While finding work, each worker maintains a pointer to the youngest node that has to be kept for the benefit of other workers. Such pointers are used and updated by the `Allocate...Node` functions. When an assignment is started (`Found...Work`) the top of stack pointers for the other WAM stacks are initialised from relevant fields of the node physically preceding the embryonic node of the new assignment, as these fields define how much of the other stacks has to be kept.

8.4 Pruning Operators

Pruning operations must distinguish between (i) pruning local nodes only, (ii) pruning remote nodes, and (iii) pruning public nodes. In cases (i) and (ii), the node can be pruned right away, but the memory occupied by the pruned node can only be reclaimed in case (i). The trail must be tidied in all three cases, as explained in [11]. In case (iii), the scheduler is responsible for pruning the public nodes, but may decide to suspend or abort the current assignment instead, forcing the engine to invoke `Sched_Suspend` or `Sched_Be_Pruned`, respectively. Note that `Sched_Prune` is invoked in all three cases, to give the scheduler an opportunity to keep pruning information up to date.

To support suspension of cuts and commits, the compiler provides extra information about what temporary variables need to be saved until the suspended task is resumed. This extra information also encodes the type of the pruning operator.

8.5 Premature Termination

To suspend the current assignment when the scheduler uses the "suspend" exit in `Sched_Prune`, `Sched_Synch`, or `Sched_Check`, the engine creates an auxiliary node which stores the current state of computation and calls `Sched_Suspend`. It is up to the scheduler to decide when the suspended work may be resumed.

To abort the current assignment when the scheduler uses the "be_pruned" exit in the above functions, the engine deinstalls all conditional bindings made by the current assignment, marks all remote nodes as reclaimable except the sentry node, and calls `Sched_Be_Pruned`.

8.6 Movement

While executing Prolog code, the binding array is kept in phase with the trail stack: whenever a binding is added to or removed from the trail, the bound value is also stored or erased in the binding array. While finding work, the engine maintains a pointer to a node in the tree corresponding to the current contents of the binding array. When the scheduler asks the engine to "move" the binding array up to a new position (`Move_Engine_Up`), bindings which were recorded on the trail path between the current and the new position are deinstalled from the binding array, and the current position is updated. Similarly, `Move_Engine_Down` installs a number of trailed binding in the binding array and updates the current position.

When an assignment is started (`Found...Work`), the engine positions its binding array at the tip node of the new or resumed branch in order to get ready to start executing the Prolog code.

9 Applying the Interface to Andorra-I

The engine-scheduler interface has been originally designed for the Aurora or-parallel Prolog system. Its primary purpose has been to support exchangeable use of several schedulers with a single engine (i.e. the Aurora engine based on Sicstus). Recently the interface has been used to link the and-parallel engine of the Andorra-I system with the Bristol scheduler developed in the context of Aurora.

In contrast with the Sicstus engine, Andorra-I performs and-parallel execution: any goals which can be reduced without making choicepoints (so called determinate goals) are executed eagerly in parallel; a team of workers work together to exploit and-parallelism. However, when no determinate goals remain, Andorra-I behaves similarly to Prolog: it uses the leftmost goal to make a choicepoint. Moreover, the backtracking routine resembles Prolog, as well: when a goal fails, the team backtracks to the nearest choicepoint, and starts to explore the next branch. Thus, despite the and-parallel execution phase, Andorra-I and Aurora behave in exactly the same way in exploring the or-tree. From the point of view of the interface, an Andorra-I team is exactly the same as an Aurora worker.

In the Andorra-I implementation the following data structures have been introduced to support the interface. First, in a way similar to Aurora, Andorra-I requires two additional pointers for each team: one for marking the boundary between the public and the private regions of the tree, and another for storing the current binding array position. Second, a parent pointer has to be added to each node (Andorra-I originally did not require the parent pointer because of the fixed node size). The backtracking routine is modified so that engine always calls the scheduler (`Sched_Die_Back`), if it is in the public region. To simplify the implementation, Andorra-I currently does not allow a worker to work on other workers' stacks. Therefore, when a worker resumes a suspended branch which belongs to someone else, the branch has to be made public.

The main difference between Aurora and Andorra-I arises in the handling of pruning operators. According to the interface, the engine should call the scheduler whenever it executes a pruning operator (`Sched_Prune`). If the scheduler decides that the pruning cannot go ahead, the engine is required to suspend the current branch and call `Sched_Suspend` immediately. In Andorra-I, however, the pruning operator is executed during the and-parallel phase, and there might be some other goals being executed simultaneously by fellow workers in the team. When a worker needs to suspend because of the pruning operator, it has to take care of its team, i.e. inform all other workers to stop and then find new work together. In fact, even if there is only one worker in the team, it is not easy to stop the and-parallel execution phase prematurely, without slowing down the whole execution process. Therefore, we have decided to let the team carry on the and-parallel phase and suspend later, if necessary. As a special case it may happen that the computation fails after `Sched_Prune` is called. In this case, the Andorra-I engine marks the suspended node as a *cut_fail* node. Later on, when the scheduler resumes the given branch, the engine will backtrack immediately.

Preliminary performance results of the Andorra-I system are very promising [2], showing that Andorra-I is capable of exploiting or-parallelism with similar efficiency as in Aurora. The overall experience of using the interface in the Andorra-I implementation is very positive: the interface proved to be well designed and of appropriate abstraction level.

10 Performance Results

No detailed performance analysis work has been done for the new Aurora implementation yet. Preliminary measurements have been performed with the Manchester scheduler, on the benchmark suite introduced in the performance analysis of the earlier Aurora version [14]. The benchmarks are divided into three groups according to granularity: course granularity (top section in the tables), medium granularity (middle section), and fine granularity (bottom section).

Table 1 shows the running times for that benchmark suite on the first generation of Aurora (using the old interface and an engine based on Sicstus Prolog 0.3). Table 2 shows the running times for the same benchmarks in the second generation of Aurora. There is an overall improvement of up to 60% in terms of absolute speed, mostly due to the new, much faster engine. For some of the fine granularity benchmarks the relative speedups have deteriorated; this is because the increase in engine speed implies a relative increase in scheduler overheads. For benchmarks with coarse granularity, and especially for the ones with frequent suspension and resumption (e.g. `tina`), the relative speedups have improved, showing the advantages of the new interface.

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.3
8-queens1	10.11	2.54*(3.98)*	1.29*(7.84)*	0.97*(10.4)*	8.19*(1.23)*
8-queens2	29.37	7.32*(4.01)*	3.73*(7.87)*	2.76*(10.6)*	23.60*(1.24)*
tina	21.30	5.57*(3.83)*	3.02*(7.06)*	2.37*(8.98)*	17.29*(1.23)*
salt-mustard	11.71	3.03*(3.87)*	1.63*(7.18)*	1.27*(9.24)*	9.50*(1.23)*
AVERAGE		*(3.92)*	*(7.49)*	*(9.80)*	*(1.23)*
parse2 *20	9.24	2.92*(3.17)*	2.08*(4.44)*	1.96*(4.72)*	7.54*(1.23)*
parse4 *5	8.54	2.50*(3.42)*	1.67*(5.11)*	1.40*(6.10)*	6.91*(1.24)*
parse5	6.02	1.74*(3.46)*	1.17*(5.15)*	0.98*(6.14)*	4.89*(1.23)*
db4 *10	3.12	0.87*(3.60)*	0.53*(5.87)*	0.45*(6.96)*	2.69*(1.16)*
db5 *10	3.80	1.04*(3.66)*	0.64*(5.93)*	0.55*(6.92)*	3.28*(1.16)*
house *20	8.13	2.26*(3.60)*	1.40*(5.81)*	1.19*(6.84)*	6.51*(1.25)*
AVERAGE		*(3.48)*	*(5.38)*	*(6.28)*	*(1.21)*
parse1 *20	2.49	0.90*(2.77)*	0.81*(3.08)*	0.87*(2.87)*	2.02*(1.23)*
parse3 *20	2.13	0.84*(2.54)*	0.80*(2.66)*	0.83*(2.57)*	1.72*(1.24)*
farmer *100	4.83	2.34*(2.06)*	2.41*(2.00)*	2.49*(1.94)*	3.80*(1.27)*
AVERAGE		*(2.46)*	*(2.58)*	*(2.46)*	*(1.25)*

Table 1: Run times, first generation of Aurora

11 Conclusions and Future Work

We have described the engine-scheduler interface used in the second generation of the Aurora or-parallel Prolog system. We have defined a simple set of functions to cover the two basic areas of engine-scheduler interaction: finding work and communication between workers. We have identified those events during Prolog execution that may be of potential interest to schedulers, e.g. creation of nodes, entering clauses, etc. We have also developed a general characterisation of pruning properties of Prolog clauses that can be used both for scheduling speculative work and for improving the implementation of pruning operators.

The interface described in this paper is fundamentally revised with respect to earlier versions. The new interface is designed to help avoid scheduling overheads, to make the set of basic concepts simpler and more uniform, to give scope for potential optimisations including better memory management, improved treatment of pruning operations, and avoidance of speculative work.

The main purpose of the interface is to enable different engines and schedulers to be used interchangeably. To date, four separate schedulers have been written and connected to two different engines by means of the interface. Perhaps more importantly, the evolution of the interface has helped clarify many issues in implementing or-parallelism in Prolog, such as contraction and handling of pruning information.

The interface has contributed to the overall improvement of Aurora performance. We also believe that the new interface has played a significant part in the good performance results of the Bristol

Goals * repetitions	Aurora Workers 1	4	8	11	Sicstus 0.6
8-queens1	8.01	2.03 *(3.95)*	1.03 *(7.75)*	0.76 *(10.6)*	6.77 *(1.18)*
8-queens2	20.63	5.25 *(3.93)*	2.64 *(7.81)*	1.93 *(10.7)*	16.45 *(1.25)*
tina	18.40	4.65 *(3.96)*	2.39 *(7.69)*	1.79 *(10.3)*	13.78 *(1.34)*
salt-mustard	10.89	2.82 *(3.86)*	1.48 *(7.36)*	1.11 *(9.86)*	8.85 *(1.23)*
AVERAGE		*(3.92)*	*(7.65)*	*(10.4)*	*(1.25)*
parse2 *20	7.16	2.40 *(2.99)*	1.71 *(4.18)*	1.64 *(4.37)*	5.87 *(1.22)*
parse4 *5	6.67	1.85 *(3.60)*	1.40 *(4.76)*	1.19 *(5.60)*	5.40 *(1.24)*
parse5	4.71	1.42 *(3.33)*	0.96 *(4.89)*	0.81 *(5.81)*	3.82 *(1.23)*
db4 *10	2.94	0.81 *(3.63)*	0.46 *(6.39)*	0.38 *(7.82)*	2.24 *(1.31)*
db5 *10	3.56	0.97 *(3.67)*	0.57 *(6.25)*	0.47 *(7.64)*	2.73 *(1.30)*
house *20	5.07	1.47 *(3.46)*	0.93 *(5.48)*	0.79 *(6.42)*	4.22 *(1.20)*
AVERAGE		*(3.45)*	*(5.32)*	*(6.28)*	*(1.25)*
parse1 *20	1.89	0.76 *(2.47)*	0.73 *(2.61)*	0.78 *(2.42)*	1.57 *(1.20)*
parse3 *20	1.62	0.72 *(2.24)*	0.68 *(2.37)*	0.72 *(2.25)*	1.34 *(1.21)*
farmer *100	3.61	1.92 *(1.88)*	2.13 *(1.69)*	2.19 *(1.65)*	3.06 *(1.18)*
AVERAGE		*(2.20)*	*(2.22)*	*(2.11)*	*(1.20)*

Table 2: Run times, second generation of Aurora

scheduler. The Bristol scheduler has been designed with the new interface in mind, and, in spite of applying a very simple scheduling strategy, its performance is comparable (and sometimes better than) that of the earlier schedulers [3].

The main outstanding issue which has not been treated in the interface is garbage collection. Patrick Weemeeuw [17] has addressed the problem of garbage collection of the public parts of the tree. Since such activities involve synchronisation between workers and possibly relocation of scheduler data, it is likely that the interface will have to be extended to support garbage collection.

The interface has recently been utilised in a project based on the Muse approach to or-parallel Prolog [1]. An or-parallel version of BIM_Prolog [4] is currently being produced by modifying the BIM engine and connecting it via the interface to the Muse scheduler.

We are convinced that the applicability of the interface extends beyond or-parallel Prolog systems. The Andorra experience is powerful evidence of this fact, but it must be stressed that in this case, the interface was used to add or-parallelism to an already and-parallel system. Generalising the interface to cover issues of and-or-parallel scheduling could be an interesting research direction to be pursued in the future.

12 Acknowledgements

The work on engine-scheduler interfaces was initiated by David Warren. Earlier versions of the interface were developed by Ross Overbeek and Alan Calderwood. The design of the new interface benefited from several discussions with Tony Beaumont, Per Brand, Bogumił Hausman and Ewing Lusk.

The authors are indebted to Feliks Kluźniak, Ewing Lusk, and the anonymous referees for careful reading and valuable comments on drafts of this paper.

This work was supported by ESPRIT projects 2471 ("PEPMA") and 2025 ("EDS").

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse approach to or-parallel Prolog. *International*

Journal of Parallel Programming, 19(2):129–162, April 1990.

[2] Anthony Beaumont, S. Muthu Raman, Vítor Santos Costa, Péter Szeredi, David H. D. Warren, and Rong Yang. Andorra-I: An implementation of the Basic Andorra Model. Technical Report TR-90-21, University of Bristol, Computer Science Department, September 1990. Presented at the Workshop on Parallel Implementation of Languages for Symbolic Computation, July 1990, University of Oregon.

[3] Anthony Beaumont, S. Muthu Raman, and Péter Szeredi. Flexible scheduling or-parallelism in Aurora: the Bristol scheduler. In *PARLE 91, Conference on Parallel Architectures and Languages Europe*. Springer-Verlag, June 1991.

[4] BIM. BIM_Prolog release 2.4. 3078 Everberg, Belgium, March 1989.

[5] Per Brand. Wavefront scheduling. Internal Report, Gigalips Project, 1988.

[6] Ralph Butler, Terry Disz, Ewing Lusk, Robert Olson, Ross Overbeek, and Rick Stevens. Scheduling OR-parallelism: an Argonne perspective. In *Proceedings of the Fifth International Conference on Logic Programming*, pages 1590–1605. MIT Press, August 1988.

[7] Alan Calderwood. Aurora—description of scheduler interfaces. Internal Report, Gigalips Project, January 1988.

[8] Alan Calderwood and Péter Szeredi. Scheduling or-parallelism in Aurora—the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.

[9] Mats Carlsson and Péter Szeredi. The Aurora abstract machine and its emulator. SICS Research Report R90005, Swedish Institute of Computer Science, 1990.

[10] Bogumił Hausman. *Pruning and Speculative Work in OR-Parallel PROLOG*. PhD thesis, The Royal Institute of Technology, Stockholm, 1990.

[11] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[12] Vítor Santos Costa, David H. D. Warren, and Rong Yang. Andorra-I: A parallel Prolog system that transparently exploits both and- and or-parallelism. In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM Press, April 1991.

[13] Raed Sindaha. Scheduling speculative work in the Aurora or-parallel Prolog system. Internal Report, Gigalips Project, March 1990.

[14] Péter Szeredi. Performance analysis of the Aurora or-parallel Prolog system. In *Proceedings of the North American Conference on Logic Programming*, pages 713–732. MIT Press, October 1989.

[15] Péter Szeredi and Mats Carlsson. The engine-scheduler interface in the Aurora or-parallel Prolog system. Technical Report TR-90-09, University of Bristol, Computer Science Department, April 1990.

[16] David H. D. Warren. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[17] Patrick Weemeeuw. Memory compaction for shared memory multiprocessors, design and specification. In *Proceedings of the North American Conference on Logic Programming*. MIT Press, October 1990.